Terms used **premature shutdown** and **JVM** Found **78** of **134,837**

Sort results by: relevance
Display results: expanded form
Save results to a Binder
Search Tips
Open results in a new window
Try an Advanced Search
Try this search in The ACM Guide

Results 1 - 20 of 78 Result page: **1** 2 3 4 next

Relevance scale

1 Core semantics of multithreaded Java
Jeremy Manson, William Pugh
June 2001 **Proceedings of the 2001 joint ACM-ISCOPE conference on Java Grande**
Full text available: pdf(816.27 KB) Additional Information: full citation, abstract, references, citings, index terms

Java has integrated multithreading to a far greater extent than most programming languages. It is also one of the only languages that specifies and requires safety guarantees for improperly synchronized programs. It turns out that understanding these issues is far more subtle and difficult than was previously thought. The existing specification makes guarantees that prohibit standard and proposed compiler optimizations; it also omits guarantees that are necessary for safe execution of much ex ...

2 The Jrpm system for dynamically parallelizing Java programs
Michael K. Chen, Kunle Olukotun
May 2003 **ACM SIGARCH Computer Architecture News , Proceedings of the 30th annual international symposium on Computer architecture**, Volume 31 Issue 2
Full text available: pdf(320.42 KB) Additional Information: full citation, abstract, references

We describe the Java runtime parallelizing machine (Jrpm), a complete system for parallelizing sequential programs automatically. Jrpm is based on a chip multiprocessor (CMP) with thread-level speculation (TLS) support. CMPs have low sharing and communication costs relative to traditional multiprocessors, and thread-level speculation (TLS) simplifies program parallelization by allowing us to parallelize optimistically without violating correct sequential program behavior. Using a Java virtual ma ...

3 A formal specification of Java class loading
Zhenyu Qian, Allen Goldberg, Alessandro Coglio
October 2000 **ACM SIGPLAN Notices , Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications**, Volume 35 Issue 10
Full text available: pdf(241.45 KB) Additional Information: full citation, abstract, references, citings, index terms

The Java Virtual Machine (JVM) has a novel and powerful mechanism to support lazy, dynamic class loading according to user-definable policies. Class loading directly impacts type safety, on which the security of Java applications is based. Conceptual bugs in the loading mechanism were found in earlier versions of the JVM that lead to type violations. A deeper understanding of the class loading mechanism, through such means as formal

analysis, will improve our confidence that no additional bugs a ...

4 An introduction to openssl programming, part I of II

Eric Rescorla
September 2001 **Linux Journal**, Volume 2001 Issue 89

Full text available: html(27.77 KB) Additional Information: full citation, abstract, index terms

Filling in the gaps of the OpenSSL manual pages.

5 Implementing jalapeño in Java

Bowen Alpern, C. R. Attanasio, Anthony Cocchi, Derek Lieber, Stephen Smith, Ton Ngo, John J. Barton, Susan Flynn Hummel, Janice C. Sheperd, Mark Mergen
October 1999 **ACM SIGPLAN Notices , Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications**, Volume 34 Issue 10

Full text available: pdf(1.57 MB) Additional Information: full citation, abstract, references, citings, index terms

Jalapeño is a virtual machine for Java™ servers written in Java.A running Java program involves four layers of functionality: the user code, the virtual-machine, the operating system, and the hardware. By drawing the Java / non-Java boundary below the virtual machine rather than above it, Jalapeño reduces the boundary-crossing overhead and opens up more opportunities for optimization.To get Jalapeño started, a boot image of a ...

6 Targeting GNAT to the Java virtual machine

Cyrille Comar, Gary Dismukes, Franco Gasperoni
November 1997 **Proceedings of the conference on TRI-Ada '97**

Full text available: pdf(1.72 MB) Additional Information: full citation, references, citings, index terms

7 Combining Ada 95, Java byte code, and the distributed systems annex

Brad Balfour
November 1997 **Proceedings of the conference on TRI-Ada '97**

Full text available: pdf(1.75 MB) Additional Information: full citation, index terms

8 The apprentice challenge

J. Strother Moore, George Porter
May 2002 **ACM Transactions on Programming Languages and Systems (TOPLAS)**, Volume 24 Issue 3

Full text available: pdf(212.09 KB) Additional Information: full citation, abstract, references, citings, index terms

We describe a mechanically checked proof of a property of a small system of Java programs involving an unbounded number of threads and synchronization, via monitors. We adopt the output of the javac compiler as the semantics and verify the system at the bytecode level under an operational semantics for the JVM. We assume a sequentially consistent memory model and atomicity at the bytecode level. Our operational semantics is expressed in ACL2, a Lisp-based logic of recursive functions. Our proofs ...

Keywords: Java, Java Virtual Machine, mutual exclusion, operational semantics, parallel and distributed computation, theorem proving

9 Tuning garbage collection for reducing memory system energy in an embedded java environment

G. Chen, R. Shetty, M. Kandemir, N. Vijaykrishnan, M. J. Irwin, M. Wolczko

November 2002 **ACM Transactions on Embedded Computing Systems (TECS)**, Volume 1 Issue 1

Full text available: pdf(740.23 KB) Additional Information: full citation, abstract, references, index terms

Java has been widely adopted as one of the software platforms for the seamless integration of diverse computing devices. Over the last year, there has been great momentum in adopting Java technology in devices such as cellphones, PDAs, and pagers where optimizing energy consumption is critical. Since, traditionally, the Java virtual machine (JVM), the cornerstone of Java technology, is tuned for performance, taking into account energy consumption requires reevaluation, and possibly redesign of t ...

Keywords: Garbage collector, Java Virtual Machine (JVM), K Virtual Machine (KVM), low power computing

10 The mutual exclusion problem: partII—statement and solutions

Leslie Lamport

April 1986 **Journal of the ACM (JACM)**, Volume 33 Issue 2

Full text available: pdf(1.83 MB) Additional Information: full citation, abstract, references, citings, index terms, review

The theory developed in Part I is used to state the mutual exclusion problem and several additional fairness and failure-tolerance requirements. Four "distributed" N-process solutions are given, ranging from a solution requiring only one communication bit per process that permits individual starvation, to one requiring about N! communication bits per process that satisfies every reasonable fairness and failure-tolerance requirement that we can c ...

11 Integration and applications of the TAU performance system in parallel Java environments

Sameer Shende, Allen D. Malony

June 2001 **Proceedings of the 2001 joint ACM-ISCOPE conference on Java Grande**

Full text available: pdf(2.17 MB) Additional Information: full citation, abstract, references, citings, index terms

Parallel Java environments present challenging problems for performance tools because of Java's rich language system and its multi-level execution platform combined with the integration of native-code application libraries and parallel runtime software. In addition to the desire to provide robust performance measurement and analysis capabilities for the Java language itself, the coupling of different software execution contexts under a uniform performance model needs careful consideration of ...

12 Secure and portable database extensibility

Michael Godfrey, Tobias Mayr, Praveen Seshadri, Thorsten von Eicken

June 1998 **ACM SIGMOD Record , Proceedings of the 1998 ACM SIGMOD international conference on Management of data**, Volume 27 Issue 2

Full text available: pdf(1.60 MB) Additional Information: full citation, abstract, references, citings, index terms

The functionality of extensible database servers can be augmented by user-defined functions (UDFs). However, the server's security and stability are concerns whenever new code is incorporated. Recently, there has been interest in the use of Java for database extensibility. This raises several questions: Does Java solve the security problems? How does it affect efficiency? We explore the tradeoffs involved in extending the PREDATOR object-relational database server using Java. We ...

13 Incommunicado: efficient communication for isolates
Krzysztof Palacz, Jan Vitek, Grzegorz Czajkowski, Laurent Daynas
November 2002 **ACM SIGPLAN Notices , Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications**, Volume 37 Issue 11
Full text available: pdf(386.23 KB) Additional Information: full citation, abstract, references

Executing computations in a single instance of safe language virtual machine can improve performance and overall platform scalability. It also poses various challenges. One of them is providing a fast inter-application communication mechanism. In addition to being efficient, such a mechanism should not violate any functional and non-functional properties of its environment, and should also support enforcement of application-specific security policies. This paper explores the design and implement ...

Keywords: application isolation, inter-application communication

14 A Java fork/join framework
Doug Lea
June 2000 **Proceedings of the ACM 2000 conference on Java Grande**
Full text available: pdf(732.64 KB) Additional Information: full citation, references, citings, index terms

15 Pretenuring for Java
Stephen M. Blackburn, Sharad Singhai, Matthew Hertz, Kathryn S. McKinely, J. Eliot B. Moss
October 2001 **ACM SIGPLAN Notices , Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications**, Volume 36 Issue 11
Full text available: pdf(389.44 KB) Additional Information: full citation, abstract, references, citings, index terms

Pretenuring can reduce copying costs in garbage collectors by allocating long-lived objects into regions that the garbage collector with rarely, if ever, collect. We extend previous work on pretenuring as follows. (1) We produce pretenuring advice that is neutral with respect to the garbage collector algorithm and configuration. We thus can and do combine advice from different applications. We find that predictions using object lifetimes at each allocation site in Java prgroams are accurate, whi ...

16 *AdJava*: automatic distribution of Java applications
Mohammad M. Fuad, Michael J. Oudshoorn
January 2002 **Australian Computer Science Communications , Proceedings of the twenty-fifth Australasian conference on Computer science - Volume 4**, Volume 24 Issue 1
Full text available: pdf(1.27 MB) Additional Information: full citation, abstract, references, index terms

The majority of the world's computing resources remains idle most of the time. By using this resource pool, an individual computation may be completed in a fraction of time required to run the same computation on a single machine. However, distributing a program over a number of machines proves to be a tedious and difficult job. This paper introduces a system, called AdJava, which harnesses the computing power of these under-utilized heterogeneous computers by automatically distributing the user ...

Keywords: distributed programming, software agents.

17 Proof linking: modular verification of mobile programs in the presence of lazy, dynamic linking

Philip W. L. Fong, Robert D. Cameron

October 2000 **ACM Transactions on Software Engineering and Methodology (TOSEM)**, Volume 9 Issue 4

Full text available: pdf(233.60 KB) Additional Information: full citation, abstract, references, index terms, review

Although mobile code systems typically employ link-time code verifiers to protect host computers from potentially malicious code, implementation flaws in the verifiers may still leave the host system vulnerable to attack. Compounding the inherent complexity of the verification algorithms themselves, the need to support lazy, dynamic linking in mobile code systems typically leads to architectures that exhibit strong interdependencies between the loader, the verifier, and the linker. To simp ...

Keywords: Java, correctness conditions, dynamic linking, mobile code, modularity, proof linking, safety, verification protocol, virtual machine architecture

18 A parallel, incremental and concurrent GC for servers

Yoav Ossia, Ori Ben-Yitzhak, Irit Goft, Elliot K. Kolodner, Victor Leikehman, Avi Owshanko

May 2002 **ACM SIGPLAN Notices , Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation**, Volume 37 Issue 5

Full text available: pdf(231.80 KB) Additional Information: full citation, abstract, references, citings, index terms

Multithreaded applications with multi-gigabyte heaps running on modern servers provide new challenges for garbage collection (GC). The challenges for "server-oriented" GC include: ensuring short pause times on a multi-gigabyte heap, while minimizing throughput penalty, good scaling on multiprocessor hardware, and keeping the number of expensive multi-cycle fence instructions required by weak ordering to a minimum. We designed and implemented a fully parallel, incremental, mostly concurrent colle ...

Keywords: JVM, Java, concurrent garbage collection, garbage collection, incremental garbage collection, weak ordering

19 Poster Session: Execution of monolithic Java programs on large non-dedicated collections of commodity workstations

Michael Factor, Assaf Schuster, Konstantin Shagin

November 2002 **Proceedings of the 2002 joint ACM-ISCOPE conference on Java Grande**

Full text available: pdf(127.43 KB) Additional Information: full citation, abstract, index terms

Today's typical computational environment consists of a large numbers of commodity workstations interconnected by high bandwidth networks. However, only a small portion of workstation's capacity is utilized since a large share of workstations may be idle at any given moment. Moreover, the communication subsystem dedicated to the idle workstations may be also unused. This results in immense aggregate waste of computational and network resources. We propose the general design of a runtime that exe ...

Keywords: distributed shared memory, fault tolerance, high-level programming paradigm, non-dedicated environment, utilization of idle resources

20 Reducing transfer delay using Java class file splitting and prefetching

Brad Calder, Chandra Krintz, Urs Hölzle

October 1999 **ACM SIGPLAN Notices , Proceedings of the 14th ACM SIGPLAN**

conference on Object-oriented programming, systems, languages, and applications, Volume 34 Issue 10

Full text available: pdf(1.77 MB) Additional Information: full citation, abstract, references, citings, index terms

The proliferation of the Internet is fueling the development of mobile computing environments in which mobile code is executed on remote sites. In such environments, the end user must often wait while the mobile program is transferred from the server to the client where it executes. This downloading can create significant delays, hurting the interactive experience of users.We propose Java class file splitting and class file prefetching optimizations in or ...

Results 1 - 20 of 78 Result page: **1** 2 3 4 next

h c g e cf c

L31: Entry 2 of 2 File: USPT Mar 13, 2001

DOCUMENT-IDENTIFIER: US 6202208 B1
**** See image for Certificate of Correction ****
TITLE: Patching environment for modifying a Java virtual machine and method

Abstract Text (1):
The invention includes a patch environment for a modifying a program executed by a Java Virtual Machine ("JVM") while the program is being executed. The patch environment has a patch data structure defined on an electronic memory of the computer. The patch data structure has at least one Java patch for modifying a loader environment of the JVM. A plurality of data items contained in a data structure defined on the electronic memory of the computer represents each patch of the patch data structure. A second data item is contained in a second data structure defined on the electronic memory of the computer, the data item representing each applied patch of the patch data structure that modifies the loader environment of the JVM. The method of the present invention applies an ordered set of changes to a Java program while running under the control of a Java Virtual Machine having a loader environment which manages the program's loaded software. A patch environment is created such that the patch environment can alter the loader environment of the JVM. A patch file is generated containing a change to be applied to the loaded software and is loaded into the patch environment. The patch is then applied to the loaded software by changing the loader environment of the JVM while the Java program is running.

Brief Summary Text (2):
The present invention relates in general to modifying an executing computer program, and in particular to a method and system for modifying a Java Virtual Machine ("JVM") with a patch environment.

Brief Summary Text (4):
Large-scale, complex computer systems, are brought into use through integration of software programs with a hardware platform. Simply put, the software program is a detailed plan or procedure for solving a problem, that is executed on a hardware platform. The hardware platform includes a microprocessor or microprocessors and associated support circuits such as electronic memory (such as random-access-memory or RAM, hard disk memory, or card memory) and input/output port circuits so that information can be passed between components of the system and users.

Brief Summary Text (14):
Accordingly, provided is an apparatus and method to implement controlled incremental changes to a Java software application while being executed. The controlled incremental changes to a Java Virtual Machine (JVM), which operates to execute the Java software application, is made through a patch environment. The patch environment may implement the changes while the program is executing on the host computer, or without reloading the entire Java software application.

Drawing Description Text (3):
FIG. 1 is a block diagram illustrating a Java Virtual Machine ("JVM"); and

Drawing Description Text (4):
FIG. 2 is a block diagram illustrating a patch environment of the present invention for modifying a loader environment of the JVM;

Drawing Description Text (5):
FIG. 3 is a block diagram illustrating the patch environment of the present invention for modifying a method body of the loader environment of the JVM;

Drawing Description Text (6):
FIG. 4 is a block diagram illustrating the application of a patch to modify a method body of the loader environment of the JVM;

Drawing Description Text (7):
FIG. 5 is a block diagram illustrating the application of a patch to add a method body to the loader environment of the JVM; and

Drawing Description Text (8):
FIG. 6 is a block diagram illustrating synchronization of the application of a patch to modify a method body of the loader environment of the JVM in synchrony with execution of other code of the JVM.

Detailed Description Text (2):
The Java language is computer architecture neutral and portable. Java programs are compiled to an architecture neutral byte-code format, allowing a Java application to run on any computer platform as either a stand-alone program or as a program fully integrated with an operating system used by the computer platform, as long as that platform implements the Java Virtual Machine ("JVM"). Examples of such computer platforms are personal computers ("PC"), Macintosh computers, and Unix workstations. The operating systems, which are programs responsible for controlling the allocation and usage of hardware resources such as memory, central processing unit ("CPU") time, disk space, and peripheral devices, come in numerous varieties such as Unix, Windows 98, Windows NT for PCs and PowerPC Macintosh. Java programs can run on any of these operating systems.

Detailed Description Text (3):
The JVM has a clear mission: to run one Java application. When a Java application starts, a runtime instance is spawned. Each Java application runs inside its own JVM.

Detailed Description Text (4):
Referring to FIG. 1, illustrated is a JVM 100. The JVM 100 includes a main memory 102 with a heap 104, and a JVM internal memory 106. The main memory 102 is further partitioned to define a working memory 108. The main memory 102 is the environment in which a Java program runs. The JVM 100 also includes a function component 110 for providing a garbage collection function, a system interface, a loader environment, an execution engine, and the like, including threads defined by the architecture of the JVM 100, discussed below.

Detailed Description Text (5):
Generally, a thread is a process that is part of a larger process or Java program. The conventional JVM specification establishes a threading model that seeks to facilitate implementation on a wide variety of computer and software architectures. The Java threading model allows implementation designers to use native threads (threads belonging to underlying operating systems). Alternatively, designers can implement a thread mechanism as part of their JVM implementation. An advantage to using native threads on a multi-processor host is that different threads of a Java application can run simultaneously on different CPUs.

Detailed Description Text (6):
Threads can be either daemon and non-daemon. A daemon thread is a thread used by

the JVM 100, such as a thread that performs garbage collection, discussed later in detail. The Java application, however, can mark any threads it creates as daemon threads. The initial thread of an application--the one that begins at main( )--is a non-daemon thread.

Detailed Description Text (7):
The JVM continues to exist as long as any non-daemon thread is running. When all non-daemon threads of a Java application terminate, the JVM instance will exit.

Detailed Description Text (8):
According to the JVM specification, the thread implementation of any JVM must support two aspects of synchronization: object locking, and thread wait-and-notify. Object locking helps keep threads from interfering with one another while working independently on shared data. Thread wait-and-notify helps threads to cooperate with one another while working together toward some common goal. Running-applications access the JVM locking capabilities via the instruction set, and its wait-and-notify capabilities via the wait( ), notify( ), and notifyAll( ) methods of class.

Detailed Description Text (9):
When the JVM 100 runs a Java program, memory is needed to store Java components, such as bytecodes and other information extracted from a loaded class file, objects the program instantiates, parameters to Java methods, return values, local variables, and intermediate results of computations.

Detailed Description Text (10):
Under the JVM Specification, the behavior of Java threads is defined in terms of variables, main memory, and working memory. The JVM Specification is a commercially-available document under the title "The Java Virtual Machine" by Tim Lindholm and Frank Yellin (1997), ISBN 0-201-63452-X, available from Sun Microsystems, Inc. or at Internet address http://www.aw.com/cp/javaseries.

Detailed Description Text (11):
The main memory 102, contains the Java program variables: instance variables of objects, components of arrays, and class variables. Each thread has a working memory 108, in which a JVM thread stores "working copies" of variables the thread uses or assigns. Local variables and parameters, because they are private to individual threads, can be logically seen as part of either the working memory 108 or the main memory 102.

Detailed Description Text (12):
When a class instance or array is created in a running Java application, the memory for the new object is allocated from the heap 104, which is in the portion of memory defined as the main memory 102. Because only one heap 104 exists inside a JVM 100, all threads share the heap 104. Also, because a Java application runs inside its "own" exclusive JVM instance, a separate heap 104 exists for every individual running application. In this manner, two different Java applications cannot corrupt the heap data of the other. However, two different threads of the same application could trample on the heap data of the other. For this reason, proper synchronization of multithreaded access to objects (heap data) in Java programs needs to be addressed.

Detailed Description Text (13):
The JVM 100 includes an instruction that allocates memory on the heap 104 for a new object but includes no instruction for freeing that memory. The JVM 100 is responsible for deciding whether and when to free memory occupied by objects that are no longer referenced by the running application. Usually, a JVM 100 uses a garbage collector thread to manage the heap 104.

Detailed Description Text (14):

The primary function of the garbage collector thread is to automatically reclaim the memory used by objects that are no longer referenced by the running-application. The garbage collector can also move objects as an application runs to reduce fragmentation of the heap 104. Fragmentation is the scattering of parts of an object in different areas of the heap 104, resulting in slower access and degradation of overall performance of memory operations.

Detailed Description Text (15):
A garbage collector is not strictly required by the JVM specification. The specification requires only that an implementation manage its heap 104 in some manner. For example, an implementation could simply have a fixed amount of heap space available and throw an OutOfMemory exception when that space fills up.

Detailed Description Text (16):
No garbage-collection technique is dictated by the JVM specification. Java designers can use whatever techniques seem most appropriate given their goals, constraints, and talents. Because program references to objects can exist in many places--Java Stacks, the heap, the loader environment, native method stacks--the choice of garbage-collection techniques heavily influences the design of the runtime data areas of an implementation.

Detailed Description Text (17):
A JVM 100 starts executing its solitary application by invoking the main( ) method of some initial class in a class file 124. The term "class" as used herein means a generalized category that describes a group of more specific methods that can exist within it, and are comparable in concept to the types of "pigeonholes" used to organize information. The term "method" as used herein means a procedure or a function. The data and methods, taken together, usually serve to define the contents and capabilities of some kind of object. Any class with such a main( ) method can be used as the starting point for a Java application. The main( ) method serves as the starting point for the initial thread of the application. The initial thread can in turn generate other threads.

Detailed Description Text (18):
The JVM 100 includes an execution engine which is a part of the object 110 and is a mechanism responsible for executing the instructions contained in the methods of loaded classes.

Detailed Description Text (19):
As shown in FIG. 1, the JVM internal memory 106 includes a loader environment 200 for loading types, i.e., classes and interfaces, having fully qualified names. The loader environment 200 is configured for storing metadata describing attributes about data types (or classes, such as data objects which go into the heap 104 and into the working memory 108) derived from the program and loaded through the class files and loader mechanism. These areas are shared by all threads running inside the JVM 100. When the JVM 100 loads a class file 124. the JVM 100 parses information about a "type" from the binary data contained in the class file 124. It places this type information into the loader environment 100. As the Java program runs, the JVM 100 places all objects the program initiates into the heap 104.

Detailed Description Text (20):
The JVM 100 computes by performing operations on data types. Both the data types and operations are strictly defined by the JVM Specification. The data types can be divided into a set of primitive types and a reference type. Variables of the primitive types hold primitive values, and variables of the reference type hold reference values. Reference values refer to objects but are not objects themselves. Primitive values, by contrast, do not refer to anything. They are the actual data themselves.

Detailed Description Text (21):

All the primitive types of the Java programming language, except boolean, are primitive types of the JVM 100. When a compiler translates Java source code into method bodies, it uses integers or bytes to represent booleans.

Detailed Description Text (22):
The primitive types of Java programming language other than boolean form the numeric types of the JVM 100. The numeric types are divided between the integral types (such as byte, short, int, long, and char) and floating-point types (such as float and double). The part of a Java Virtual Machine implementation that takes care of finding and loading types is the class loader subsystem, implemented through the loader environment 200 of the running, Java application.

Detailed Description Text (23):
Information about loaded types is stored in a logical area of memory called the loader environment 200. When the JVM 100 loads a type, it uses a class loader to locate the appropriate class file. The class loader reads in the class file 124 (a linear stream of binary data) and passes it to the JVM 100. The JVM 100 extracts information about the type from the binary data and stores the information in the loader environment 200. Memory for class (static) variables declared in the class is also taken from the loader environment 200.

Detailed Description Text (24):
The size of the loader environment 200 need not be fixed. As the Java application runs, the JVM 100 can expand and contract the loader environment 200 to fit the needs of the application. Implementations allow users or programmers to specify an initial size for the loader environment 200, as well as a maximum or minimum size.

Detailed Description Text (25):
The loader environment 200 can also be garbage collected by a garbage collection thread located in the function component 10. Because Java programs can be dynamically extended via class loader objects, classes can become "unreferenced" by the application. If a class becomes unreferenced, a JVM 100 can unload the class with the garbage collector thread to keep the memory occupied by the loader environment 200 at a minimum.

Detailed Description Text (26):
FIG. 2 is a block diagram representing the loader environment 200. The loader environment depicted is simplified for clarity for use in describing application and use of the invention herein. It can be readily appreciated that more complex JVM structures with larger class tables and associated memory structures can be deployed that can similarly deploy a patching environment for modifying the JVM 100 without the need to first halt the JVM.

Detailed Description Text (29):
The Java class file 124 contains everything a JVM 100 needs to know about one Java class or interface, and is set out in a precise definition of the class file format to ensure that any Java class file can be loaded and correctly interpreted by any JVM 100, no matter what computer system produced the class file 124 or what system hosts the JVM 100. The class file 124 includes a what is known as a magic number--OxCAFEBABE--that designates it as a Java file. The class file also has a version number, a constant pool 130, a method.sub.13 info portion 132, and an attributes portion 134. The constant pool 130 contains the constants associated with the class or interface defined by the file. Constants such as literal strings, final variable values, class names, and method names are stored in the constant pool 130. After the constant pool 130 is the method.sub.13 info portion 132 that contains information about a method, including the method name and descriptor (for example, the return type and argument types). If the method is not abstract, the method.sub.13 info portion 132 includes the number of memory stack words (a word having sixteen bits) required for the local methods, the maximum number of memory stack words required for the operand stack of the method, a table of exceptions

caught by the method, the method body sequence, and optional line number and local variable tables.

Detailed Description Text (30):
The last component in the class file 124 is the attributes portion 134, which gives general information about the particular class or interface defined by the class file 124. The attributes 134 has an attributes.sub.13 count field, and a count of the number of attribute_info tables appearing in the subsequent attributes list. The first item in each attributes portion 134 is an index into the constant pool 130 of a CONSTANT.sub.13 Utf8.sub.13 info table that gives the name of the attribute. Attributes come in many varieties. Several varieties are defined by the JVM Specification, discussed above, but varieties of attributes, according to well known rules, can be created and placed into the class file 124.

Detailed Description Text (35):
The data structures 204, 206, and 208 are used to allow ease of access to the method bodies B.sub.1, B.sub.2, B.sub.3, . . . B.sub.11, stored in the loader environment 200. An example of such a data structure is a method table. For each non-abstract class the JVM 100 loads into the class table 204, the JVM 100 generates a corresponding method table 208. A method table is an array of direct references to all the instance method bodies that may be invoked on a class instance.

Detailed Description Text (36):
Referring to FIG. 2, illustrated is a schematic diagram of a runtime patch environment 300 for the JVM 100. In systems that cannot be readily brought down in a controlled manner or take a large amount of time to bring down, it is preferred to modify or "patch" the program while the JVM is loaded onto its platform and operational to avoid the time and expense otherwise required.

Detailed Description Text (37):
An empty patch environment is part of the patch-capable JVM 100, and is acted on by the JVM 100 to create the patch environment 300, which is related to the loader environment 200. It should be noted that discrete memory regions can be allocated to the loader environment 200 and the patch environment 300, memory regions can be shared using conventional memory management techniques, or a hybrid memory structure can employ both allocation and memory management techniques.

Detailed Description Text (38):
For the block diagram representation shown in FIG. 2, the JVM 100 has defined a patch table data structure 302 in the patch environment 300. The data structure 300 is empty and has not yet been loaded with a patch through a patch file.

Detailed Description Text (39):
FIG. 3 is a block diagram illustrating the loading of a patch in the JVM 100. The shaded regions indicate the program structure additions generated by the JVM 100 based on the patch file 304. There are two primary patching implementations that can be conducted through the use of the patch file 304. First, altering an existing method body with a patch. Second, adding a method body with a patch.

Detailed Description Text (41):
The changes are made to Java classes that are set out in the class table 204 of the loader environment 200. After making the desired changes to one or more Java classes, a patch file 304 is generated. The patch file 304 is sufficiently similar to the class file to generate a JVM patch structure similar to that in the loader environment 200. That is, the JVM 100, based on the patched-method information in the patch file 304, generates a patch object data structure 306, a patched-method table data structure 308, and a patched-method bodies data structure 310. The JVM 100 loads, in respective data structures, a patch class 302a, a class table entry 302a, a class object 306a, a patched method table entry 308a, and patched method

bodies 310a and 310b. Accordingly, the patch class 302a has a pointer ADDR(T.sub.1 ') to the patch object T.sub.1 '. The patch object T.sub.1 ' refers to the patched method table 308. The patched method table 308 contains pointers ADDR(T', B.sub.1 ') and ADDR(T.sub.1 ', B.sub.2 ') to the patched method bodies 310a and 310b, accordingly. It should be noted that the use of the prime notation "'" indicates a Java patch that to a pre-loaded portion of a Java component in the JVM 100.

Detailed Description Text (45):
Referring to FIG. 6, in the alteration of the method bodies, the application of the patch is synchronized with respect to the execution of other code in the JVM 100 so that no interruption of function or service is necessary to apply the patch. Also, the old method bodies are retained so that the patched class can be removed by re-installing the original method bodies by reversing the swap of the pointers to the method bodies.

CLAIMS:

1. A patch environment within a Java Virtual Machine ("JVM") executing on a computer, the JVM having a loader environment containing information about software objects loaded on the computer through the JVM, the patch environment comprising:

a patch data structure defined on an electronic memory of the computer, having at least one Java patch for modifying the loader environment of the JVM, wherein the patch data structure comprising at least one method table and at least one associated method body;

at least one data item contained in the patch data structure defined on the electronic memory of the computer, identifying each patch of said patch data structure; and

a second data item associated with each of the identified patches contained in the patch data structure defined on the electronic memory of the computer for modifying the loader environment of the JVM.

4. The patch environment of claim 1 wherein said patch modifies a method body in the loader environment of the JVM by altering a method body in the loader environment.

5. The patch environment of claim 1 wherein said patch modifies a method body in the loader environment of the JVM by adding a method body in the loader environment.

6. A method of applying an ordered set of changes to an executing Java program on a computer, without interrupting the execution of the Java program, the method comprising the steps of:

(a) providing a Java Virtual Machine ("JVM") in a memory of the computer, the JVM having a loader environment wherein the Java program is loaded for execution;

(b) executing the Java program;

(c) creating a patch environment in a memory of the computer such that the patch environment can interact with the loader environment;

(d) generating a patch file having a patch;

(e) loading the patch file into the patch environment, such that the loaded patch file comprising at least one patched method table entry and at least one associated patched method body, said at least one patched method table entry having a pointer directed to said at least one associated patched method body; and

(f) applying the patch to the executing Java program.

9. A method of applying an ordered set of changes to an executing Java class file on a computer, without interrupting the execution of the Java class file, the method comprising the steps of:

(a) providing a Java Virtual Machine ("JVM") in a memory of the computer, the JVM having a loader environment wherein the Java class file is loaded for execution, the Java class file having a JVM class structure comprising at least one method table and at least one associated method body, said at least one method table having a pointer directed to said at least one associated method body;

(b) executing the Java class file;

(c) creating a patch environment in a memory of the computer such that the patch environment can interact with the loader environment;

(d) generating a patch file having a patch;

(e) loading the patch file into the patch environment, the loaded patch file being substantially similar to the class file to generate a JVM patch structure similar to the JVM class structure in the loader environment; and

(f) applying the patch to the executing Java program.

10. A method of altering an executing Java class file on a computer, without interrupting the execution of the Java class file, the method comprising the steps of:

(a) providing a Java Virtual Machine ("JVM") in a memory of the computer, the JVM having a loader environment wherein the Java class file is loaded for execution, the loaded Java class file having a JVM class structure comprising at least one method table and at least one associated method body, said at least one method table having a first pointer directed to said at least one associated method body;

(b) executing the Java class file;

(c) creating a patch environment in a memory of the computer such that the patch environment can interact with the loader environment;

(d) generating a patch file having a patch;

(e) loading the patch file into the patch environment, the loaded patch file comprising at least one associated patched method table entry and at least one patched method body, said at least one patched method table entry having a second pointer directed to said at least one associated patched method body; and

(f) replacing said at least one method body with said at least one patched method body by exchanging the first pointer with the second pointer.

12. A method of adding a new program component to an executing Java class file on a computer, without interrupting the execution of the Java class file, the method comprising the steps of:

(a) providing a Java Virtual Machine ("JVM") in a memory of the computer, the JVM having a loader environment wherein the Java class file is loaded for execution, the loaded Java class file having a JVM class structure comprising at least one method table and at least one associated method body, said at least one method table having a first pointer directed to said at least one associated method body;

(b) executing the Java class file;

(c) creating a patch environment in a memory of the computer such that the patch environment can interact with the loader environment;

(d) generating a patch file having a patch;

(e) loading the patch file into the patch environment, the loaded patch file comprising at least one patched method table entry and at least one associated patched method body, said at least one patched method table entry having a second pointer directed to said at least one associated patched method body; and

(f) adding said at least one patched method body to the executing Java class file by appending the second pointer to said at least one method table.

Hit List

Clear | Generate Collection | Print | Fwd Refs | Bkwd Refs | Generate OACS

Search Results - Record(s) 1 through 1 of 1 returned.

☐ 1. Document ID: US 6557076 B1

L33: Entry 1 of 1 File: USPT Apr 29, 2003

US-PAT-NO: 6557076
DOCUMENT-IDENTIFIER: US 6557076 B1
**** See image for Certificate of Correction ****

TITLE: Method and apparatus for aggressively rendering data in a data processing system

DATE-ISSUED: April 29, 2003

INVENTOR-INFORMATION:

NAME	CITY	STATE	ZIP CODE	COUNTRY
Copeland; George Prentice	Austin	TX		
McClain; Matthew Dale	Austin	TX		

ASSIGNEE-INFORMATION:

NAME	CITY	STATE	ZIP CODE	COUNTRY	TYPE CODE
International Business Machines Corporation	Armonk	NY			02

APPL-NO: 09/ 359278 [PALM]
DATE FILED: July 22, 1999

PARENT-CASE:
CROSS REFERENCE TO RELATED APPLICATIONS The present invention is related to applications entitled METHOD AND APPARATUS FOR HIGH-CONCURRENCY CLIENT LOCKING WITH JAVA IN A DATA PROCESSING SYSTEM, Ser. No. 09/359,274, METHOD AND APPARATUS FOR MANAGING INTERNAL CACHES AND EXTERNAL CACHES IN A DATA PROCESSING SYSTEM, Ser. No. 09/359,275, METHOD AND APPARATUS FOR CACHE COORDINATION FOR MULTIPLE ADDRESS SPACES, Ser. No. 09/359,276, METHOD AND APPARATUS FOR INVALIDATING DATA IN A CACHE, Ser. No. 09/359,277, and A METHOD AND APPARATUS FOR CACHING CONTENT IN A DATA PROCESSING SYSTEM WITH FRAGMENT GRANULARITY, Ser. No. 09/359,279, all of which are filed even date hereof, assigned to the same assignee, and incorporated herein by reference.

INT-CL: [07] G06 F 12/00

US-CL-ISSUED: 711/118; 711/113, 711/133, 711/141, 711/154, 709/216, 709/217, 707/1, 707/3
US-CL-CURRENT: 711/118; 707/1, 707/3, 709/216, 709/217, 711/113, 711/133, 711/141, 711/154

h e b b g e e e f e ef b e

FIELD-OF-SEARCH: 709/216, 709/217, 709/218, 709/219, 709/203, 709/206, 709/224, 709/229, 709/238, 709/201, 709/239, 709/245, 709/246, 711/118, 711/113, 711/106, 711/133-137, 711/141, 711/144-146, 711/154-156, 711/208, 711/210, 707/1, 707/3, 707/10, 707/100

PRIOR-ART-DISCLOSED:

U.S. PATENT DOCUMENTS

PAT-NO	ISSUE-DATE	PATENTEE-NAME	US-CL
5506967	April 1996	Barajas et al.	
5530799	June 1996	Marsh et al.	395/164
5546579	August 1996	Josten et al.	
5708789	January 1998	McClure	
6044438	March 2000	Olnowich	711/130

OTHER PUBLICATIONS

Iyengar et al.; Improving Web Server Performance by Caching Dynamic Data; Proceedings on the USENIX Symposium on Internet Technologies and Systems; 1998.
Challenger et al.; A Scalable and Highly Available System for Serving Dynamic Data At Frequently Accessed Web Sites; 1998 High Performance Networking and Computing Conference; pp. 1-23.
Challenger et al.; A Scalable System For Consistently Caching Dynamic Web Data; Infocom 1999 pp 1-22.

ART-UNIT: 2157

PRIMARY-EXAMINER: Etienne; Ario

ATTY-AGENT-FIRM: Yee; Duke W. Mims, Jr.; David A.

ABSTRACT:

A method and apparatus for processing data stored in a memory. Responsive to data being invalidated in the memory, a determination is made as to whether an indicator is associated with the data. Responsive to the indicator being associated with the data, the data is retrieved from a source. The data is refreshed in the memory without requiring an external request for the data. Servers typically operate with enough spare capacity so that peek loads can be handled. Aggressive rerendering allows this spare capacity to be used to refresh pages in the cache when they have been invalidated.

40 Claims, 18 Drawing figures

(24 AND 32).USPT.	1
(L24 AND L32).USPT.	1

Display Format: FRO Change Format

Previous Page Next Page Go to Doc#

h e b b g e e e f e ef b e

Hit List

Clear | Generate Collection | Print | Fwd Refs | Bkwd Refs | Generate OACS

Search Results - Record(s) 1 through 1 of 1 returned.

☐ 1. Document ID: US 6209018 B1

L29: Entry 1 of 1 File: USPT Mar 27, 2001

US-PAT-NO: 6209018
DOCUMENT-IDENTIFIER: US 6209018 B1

TITLE: Service framework for a distributed object network system

DATE-ISSUED: March 27, 2001

INVENTOR-INFORMATION:

NAME	CITY	STATE	ZIP CODE	COUNTRY
Ben-Shachar; Ofer	Palo Alto	CA		
Anand; Vijay	Mountain View	CA		
Ebbs; Ken	Mountain View	CA		
Malka; Yarden Yaacov	Menlo Park	CA		
Brewster; David Latimer	Santa Clara	CA		

ASSIGNEE-INFORMATION:

NAME	CITY	STATE	ZIP CODE	COUNTRY	TYPE CODE
Sun Microsystems, Inc.	Palo Alto	CA			02

APPL-NO: 08/ 969982 [PALM]
DATE FILED: November 13, 1997

INT-CL: [07] G06 F 13/00

US-CL-ISSUED: 709/105
US-CL-CURRENT: 718/105

FIELD-OF-SEARCH: 709/105, 709/102, 395/675

PRIOR-ART-DISCLOSED:

U.S. PATENT DOCUMENTS

PAT-NO	ISSUE-DATE	PATENTEE-NAME	US-CL
5864866	January 1999	Henckel et al.	707/103
5872971	February 1999	Knapman et al.	709/101
5903725	May 1999	Colyer	707/103

h e b b g e e e f e ef b e

FOREIGN PATENT DOCUMENTS

FOREIGN-PAT-NO	PUBN-DATE	COUNTRY	US-CL
0 613 083	August 1994	EP	

OTHER PUBLICATIONS

Tokmakoff et al.; Service Brokering in Object-Based Systems: Advanced Information Services; Third International Workshop on Community Networking, 1996; pp. 43-48, May 1996.*
Harry M. Sneed; Encapsulating Legacy Software for Use in Client/Server Systems; Proceedings of the Third Working Conf. on Reverse Enginerring, 1996; pp. 104-119, Nov. 1996.*
Takuma Sudo et al.: "Transaction Processing in Distributed Environments" Hitachi Review, vol. 45, No. 2, Apr. 1, 1996, pp. 55-60, XP000622834.
"Reducing the Latency of Distributed Resource Registration" IBM Technical Disclosure Bulletin, vol. 38, No. 5, May 1, 1995, p. 419/420, XP000519630.
HTML Sourcebook, Third Edition, "A Completer Guide to HTML 3.2 and HTML Extensions", by Ian Graham, 1997.
"Internet Agents: Spiders, Wanderers, Brokers, and Bots", by Fah-Chun Cheong, 1996.

ART-UNIT: 214

PRIMARY-EXAMINER: Coulter; Kenneth R.

ATTY-AGENT-FIRM: Gunnison; Forrest Gunnison, McKay & Hodgson, L.L.P.

ABSTRACT:

An improved method and apparatus for providing a service framework for a distributed object network system are provided. In some embodiments, an apparatus that includes a server, a service for a limited resource residing on the server, and a pool of workers for the service that execute service requests from a client in a distributed object network system is provided. In some embodiments, a method that includes providing client-side service request encapsulation, balancing workloads among clones of service locators, clones of services, and workers in a worker pool of a service, and improving fault tolerance in a distributed object network system is provided.

16 Claims, 33 Drawing figures

Term	Documents
JAVA	7695
JAVAS	4
(28 AND JAVA).USPT.	1
(JAVA AND L28).USPT.	1

h e b b g e e e f e ef b e

Display Format: FRO Change Format

Previous Page Next Page Go to Doc#

h e b b g e e e f e ef b e

L31: Entry 1 of 2 File: USPT Aug 5, 2003

DOCUMENT-IDENTIFIER: US 6604106 B1
TITLE: Compression and delivery of web server content

Abstract Text (1):
A server-side mechanism together with an optional client-side decompression process enhance server content delivery. The server-side mechanism preferably comprises a pair of processes: a daemon process and a servlet process. The daemon process recursively compresses directories of content (HTML, graphics files, and the like) while instances of the servlet process, in parallel, serve content. When a target directory is completely compressed, the files which previously existed in an uncompressed state are either archived or deleted. The servlet process interprets the compressed objects, resolving the connection between the client and server, and serves out the requested content. If the request originates from a client that is not enabled to decompress files, the servlet decompresses the requested files on-the-fly. When supported on a given client machine, the client process decompresses the streaming content for use on the client system.

Brief Summary Text (16):
These and other objects of the invention are realized using a server-side mechanism together with an optional client-side decompression process. The server-side mechanism preferably comprises a pair of processes: a daemon process and a servlet process. The daemon process is a server-side executable that executes seamlessly and transparently to the regular operation and Web serving tasks on the host. The daemon process recursively compresses directories of content (HTML, graphics files, and the like) while the server, in parallel, serves content. When a target directory is completely compressed, the files which previously existed in an uncompressed state are either archived or deleted.

Brief Summary Text (17):
The servlet process interprets the compressed objects, resolving the connection between the client and server, and serves out the requested content. If the request originates from a client that is not enabled to decompress files, the servlet decompresses the requested files on-the-fly. The servlet also resolves connections when a client requests only partial content requested from within a compressed group of files. The daemon and servlet processes preferably operate asynchronously to each other.

Brief Summary Text (19):
Each of the processes of the present invention includes an application programming interface (API), namely, a program entry point, that allows the respective process to be extended by other software. As a result, different types of compression and their corresponding decompression routines are plug compatible with the architecture. Further, by interfacing through the API, the daemon and the servlet may readily support other markup language types (e.g., SGML, XML, HDML, and others) so that the inventive architecture is compatible with Internet appliances and other pervasive computing devices (e.g., palmtops, PDAs, cell phones, and the like) that do not include the full HTML function set.

Drawing Description Text (6):
FIG. 4 is a block diagram of the daemon process of the present invention;

Detailed Description Text (4):
A representative Web client is a personal computer that is x86-, PowerPC.RTM.- or RISC-based, that includes an operating system such as IBM.RTM. OS/2.RTM. or Microsoft Windows '95, and that includes a Web browser, such as Netscape Navigator 4.0 (or higher), having a Java Virtual Machine (JVM) and support for application plug-ins or helper applications.

Detailed Description Text (5):
The Web server accepts a client request and returns a response. The operation of the server program 22 is governed by a number of server application functions (SAFs), each of which is configured to execute in a certain step of a sequence. This sequence, illustrated in FIG. 2 by way of background only, begins with authorization translation (AuthTrans) 30, during which the server translates any authorization information sent by the client into a user and a group. If necessary, the AuthTrans step may decode a message to get the actual client request. At step 32, called name translation (NameTrans), the URL associated with the request may be kept intact or it can be translated into a system-dependent file name, a redirection URL or a mirror site URL. At step 34, called path checks (PathCheck), the server performs various tests on the resulting path to ensure that the given client may retrieve the document. At step 36, sometimes referred to as object types (ObjectType), MIME (Multipurpose Internet Mail Extension) type information (e.g., text/html, image/gif, etc.) for the given document is identified. At step 38, called Service (Service), the Web server routine selects an internal server function to send the result back to the client. This function can run the normal server service routine (to return a file), some other server function (such as a program to return a custom document) or a CGI program. At step 40, called Add Log (AddLog), information about the transaction is recorded.

Detailed Description Text (6):
FIG. 3 illustrates the main components of the invention. They include a pair of processes that execute on the server and, optionally, a process that executes on the client. The server processes comprise first process 42 and second process 44. The client process is a third process 46, which is used to manage decompression on the client. In general, first process 42 is used for compressing files on the server, and second process 44 is used to satisfy client HTTP requests. Second process 44 operates to serve compressed files if a client request initiates from a client that supports the third process 46; alternatively, second process 44 operates to decompress compressed files prior to serving the resulting content if the client request initiates from a client that does not support the third process 46.

Detailed Description Text (7):
In a representative embodiment, first process 42 is a platform executable daemon, second process 44 is a Java servlet, and third process 46 is a Java applet. As is well-known, a Java servlet or applet, as the case may be, comprises class files executable in a Java Virtual Machine (JVM). A conventional Web browser typically includes a JVM; thus, the respective servlet and applet processes are executable within the content of an existing Web server and Web client architecture. A client machine that supports the third process is sometimes referred to herein as a "plug-in enabled" client because it includes support for a given decompression routine.

Detailed Description Text (8):
According to the invention, the daemon process 42 is a server side executable that executes seamlessly and transparently to the regular operation of the Web serving tasks running on the computer. As seen in FIG. 4, the deamon process 42 includes a daemon manager 43 and a compression routine 45. In operation, the deamon manager 43 schedules files for compression by the compression routine 45 as those files are created or updated. The daemon manager 43 stores the resulting compressed files in DASD 47. In operation, manager 43 recursively navigates through the Web server

directories, compressing the content (using compression routine 45), and storing the resultant compressed binary files, preferably within their respective subdirectories. This operation occurs as the server (and, in particular, the servlet process), in parallel, is serving content. The compressed binary files may be moved from their origin directories and stored elsewhere provided the servlet has access to a mapping file from which it may obtain the location of these files.

Detailed Description Text (9):
Thus, the deamon process operates asynchronously with respect to other server operations including the servlet process. When the daemon process is finished compressing a target directory, the files that previously existed in an uncompressed state are either archived (in DASD 47) or deleted depending on a configuration setting for the process. These operations are effected by the daemon manager 43. As is well-known, a dialog panel may be used in a conventional manner to establish and/or modify those configuration settings. The particular panel interface is created by the daemon manager 43.

Detailed Description Text (10):
The daemon process 42 thus provides significant space savings on the DASD by compressing files on the server. Preferably, the daemon process is configurable to compress all directories on the server, certain directories on the server, or other selected content. Thus, for example, a given Web server may be responsible for serving one or more Web sites. In such case, the administrator may desire to compress a given directory (corresponding to a first Web site or URL) while maintaining other directories (corresponding to a second Web site or URL) uncompressed. The daemon process includes a configurable interface through which the administrator may specify particular subdirectories on the DASD 47 corresponding to each such site.

Detailed Description Text (11):
The deamon process 42 is also preferably configurable to aggregate given files into a single binary file (e.g., a zip file). In addition, the deamon process 42 may also be configured to "transcode" files from one source markup language to another markup language. This function is provided by transcode routine 41. As an example, assume it is desired to render given content originally written in HTML on a pervasive computing client (e.g., a PDA such as the IBM WorkPad.TM., a smartphone, a palmtop device, any Internet appliance, or the like) that does not support the full function set of an HTML Windows-based client. In such case, it is necessary to transcode the HTML-based file into a format (e.g., HDML or handheld device markup language) compatible with such a device so that the file may be appropriately rendered on the client. To this end, the deamon includes an API 48 (namely, a program entry point) that allows the daemon to be extended by other software. This enables the daemon to readily support other markup language types (e.g., SGML, XML, HDML, and others). In addition, different types of compression routines 45 are "plug" compatible with the daemon by interfacing through the API. This significantly enhances the flexibility of the inventive architecture.

Detailed Description Text (12):
Referring now to FIG. 5, each servlet 44 is controlled by a servlet manager 50, which is part of the existing Web server infrastructure. Whenever a new client attaches to the server, the servlet manager 50 instantiates a new service thread, thereby generating a new instance 52 of the servlet. Each servlet comprises three (3) basic routines: an init ( ) routine 54, a destroy ( ) routine 56, and a service ( ) routine 58. The init ( ) routine 54 provides initialization functionality (typically on a "one-time" basis). These are the functions that enable the servlet to be recognized and managed by the servlet manager and other server functions. The destroy ( ) routine 56 selectively destroys the servlet, which is an atypical operation. The service ( ) routine 58 is the primary engine of the servlet. In particular, the service ( ) routine is used to satisfy the GET or PUT HTTP request received from the client (and that spawned the servlet instance). This process is

now described in more detail below and in conjunction with the flowchart of FIG. 6.

Detailed Description Text (13):
The service ( ) routine provides the basic operating functionality of the servlet. As noted above, an instance of the servlet is spawned upon receipt of an HTTP transaction request (e.g., a GET or PUT) from the client. As previously described, the client may be a conventional Windows-based machine (having a full HTML feature set), or a pervasive computing device (having a less than full-feature set). The routine begins at step 60 when the servlet is spawned. At step 62, the routine initially examines the HTTP request to determine whether the client machine is plug-in enabled. If the client relies on the HTTP 401 authentication method and the browser supports cookies, step 62 simply examines the cookie. Alternative techniques may be used to determine whether the client machine has decompression functionality. If the client is plug-in enabled, the routine branches to step 64 to resolve the URL. This step identifies the specific files on the DASD that must be served in response to the request. As noted above, the daemon process typically aggregates a given Web site (comprising many files of a subdirectory) into a single file. Thus, step 64 is useful in identifying the particular files that must be served. In step 64, the servlet also resolves connections when the client requests only partial content requested from within a compressed group of files. At step 66, the compressed files are located, retrieved and served to the client. Alternatively, if the client is not plug-in enabled, the routine branches (from step 62) to step 68. In this step, the service routine resolves the URL as previously noted. The requested files are then located and retrieved in step 70. In step 72, these files are decompressed "on-the-fly" and then served in step 72. Following the AddLog function (as previously described with respect to FIG. 2), the service ( ) routine of the servlet terminates. Control is then returned back to the servlet manager.

Detailed Description Text (15):
In a given transaction, the client establishes the connection to the servlet. The servlet determines that client is requesting a particular URL (representing a compressed set of files) within a compressed file. The servlet then serves that file as a binary string. Of course, multiple instances of the servlet operate concurrently on the server under the control of the servlet manager, and this set of processes operates asynchronously with respect to the daemon process.

Detailed Description Text (21):
The invention is thus useful in optimizing storage of Web server content. Once configured and initiated, the daemon process 42 begins recursively creating compressed files (in the aggregate or separately). The daemon process (depending on its configuration settings) may then delete or archive the original files. The servlet process 44 is then started. The servlet process handles client HTTP (or other protocol) requests and dynamically decompresses data to the client (depending on whether the client is configured to handle the compressed data stream). As part of the compression step (by the servlet), preferably the servlet also includes the architected interface (namely, the API) that allows for specific types of compression. Moreover, this interface allows for a plurality of data manipulation routines to be registered with the servlet. Such routines may produce an alternate form of the data that is specifically optimized for various "pervasive" type devices.

Detailed Description Text (26):
As noted above, the above-described functionality preferably includes a server side piece and a client side piece. The server side piece comprises the daemon and servlet processes. The client side piece is the plug-in enabled code, preferably a Java applet. In either case, the functionality is implemented in software executable in a processor, namely, as a set of instructions (program code) in a code module resident in the random access memory of the computer. Until required by

the computer, the set of instructions may be stored in another computer memory, for example, in a hard disk drive, or in a removable memory such as an optical disk (for eventual use in a CD ROM) or floppy disk (for eventual use in a floppy disk drive), or downloaded via the Internet or other computer network.

CLAIMS:

1. A method for increasing effective delivery of contents supported on a server, comprising the steps of: using a first process operating on the server to compress given content into a compressed file; and as the first process is operating, using a second process operating on the server to respond to client requests for the content wherein the first process is a platform executable and the second process is a Java servlet.

2. A server, comprising: a storage device; a first process for compressing given content into a compressed file and storing the compressed file in the storage device; and a second process operating asynchronously with respect to the first process for responding to client requests for the content wherein the first process is a platform executable and the second is a Java servlet.

4. A computer program product in a computer-readable medium, comprising; a first process executable on the server for compressing given content into a compressed filed and storing the compressed file in a storage device; and a second process executable on the server and operating asynchronously with respect to the first process for responding to client request for the content wherein the first process is a platform executable and the second process is a Java servlet.

5. A computer program product in a computer-readable medium, comprising; a first process executable on the server for compressing given content into a compressed filed and storing the compressed file in a storage device; and a second process executable on the server and operating asynchronously with respect to the first process for responding to client request for the content wherein the first process also transcodes content from a first markup language to a second markup language format.

First Hit Fwd Refs

End of Result Set

Generate Collection Print

L15: Entry 1 of 1 File: USPT Dec 30, 2003

DOCUMENT-IDENTIFIER: US 6671686 B2
TITLE: Decentralized, distributed internet data management

Abstract Text (1):
A light-weight architecture is provided, where each component is in itself its own advanced mini-transaction processing monitor. To accomplish this, the system is most readily implemented as a set of Java classes. The resulting architecture is as follows. In a composite system, each server is an independent component performing its own scheduling and transaction management. These servers are built using Java and inheriting from the classes provided by the system according to the invention. The interface to each server defines the services it implements. An invocation of one of these services (through remote method invocation) results in the creation of a local transaction (child of the invoking transaction and parent of any transaction that might be triggered by invoking the services of other servers). Each transaction is a thread that (in an exemplary system) can invoke SQL statements in a local database (directly connected to that server) as well as services offered by other servers. All the information that is required to build a global composite transaction is implicitly added by the system to each call. Each transaction is, however, independently handled at each server. That is, the servers neither communicate among themselves nor rely on a centralized component to make scheduling or recovery decisions. In this way, components can be dynamically added or removed from the system without compromising correctness. All a new server needs to know is the interface and address of the servers it will invoke. Regardless of the configuration, the system according to the invention guarantees that transactions executed over these servers will be correct (serializable) and recoverable at a global and local level.

Brief Summary Text (35):
Enterprise Java Beans. This is the Java vision for distributed transaction processing applications. Enterprise Java Beans is a standard, meaning that it consists of specifications rather than implementations. The main objective is to provide a portable way of writing transactional applications. By taking all server-specific issues out of the application (such as transaction management, pooling of resources, swapping of inactive components), it is possible to create portable applications (so-called Beans, the Java terminology for a software component). The key idea is that all these components have to adhere to a standardized way of interacting with the server environment. In practice, this means that a component has a set of predefined methods that are called by the server in case of important events. For instance, before swapping out an inactive component, this component is notified by calling its method ejbPassivate(), whose implementation should discard any volatile data and synchronize the component's database state. The whole concept of this technology is thus oriented towards component-based server applications, and the contract between a component and the server can be very complex. As such, it is orthogonal to the objectives discussed here: although EJB mainly targets applications with transactional aspects, the issue of transaction management itself is left open. In that approach, some of the problems with distributed transactions are recognized, but no attempts are made to solve them. Finally, with JavaBeans, nested transactions are not currently supported.

Brief Summary Text (42):
In the system according to the invention, a light-weight architecture is provided, where each component is in itself its own advanced mini-transaction processing monitor. To accomplish this, the system is most readily implemented as a set of Java classes. The resulting architecture is as follows. In a composite system (such as that of FIG. 2), each server is an independent component performing its own scheduling and transaction management. These servers are built using Java and inheriting from the classes provided by the system according to the invention. The interface to each server defines the services it implements. An invocation of one of these services (through RMI or "remote method invocation") results in the creation of a local transaction (child of the invoking transaction and parent of any transaction that might be triggered by invoking the services of other servers). Each transaction is a thread that can, for example, invoke SQL statements in a local database (directly connected to that server) as well as services offered by other servers. All the information that is required to build a global composite transaction is implicitly added by the system to each call. Each transaction is, however, independently handled at each server. That is, the servers neither communicate among themselves nor rely on a centralized component to make scheduling or recovery decisions. In this way, components can be dynamically added or removed from the system without compromising correctness. All a new server needs to know is the interface and address of the servers it will invoke. Regardless of the configuration, the system according to the invention guarantees that transactions executed over these servers will be correct (serializable) and recoverable at a global and local level.

Detailed Description Text (2):
As described above, in the system according to the invention, a light-weight architecture is provided, where each component is in itself its own advanced mini-transaction processing monitor. To accomplish this, the system is most readily implemented as a set of Java classes. The resulting architecture is as follows. In a composite system (such as that of FIG. 2), each server is an independent component performing its own scheduling and transaction management. These servers are built using Java and inheriting from the classes provided by the system according to the invention. The interface to each server defines the services it implements. An invocation of one of these services (through RMI or "remote method invocation") results in the creation of a local transaction (child of the invoking transaction and parent of any transaction that might be triggered by invoking the services of other servers). Each transaction is, for example, a thread that can invoke SQL statements in a local database (directly connected to that server) as well as services offered by other servers. All the information that is required to build a global composite transaction is implicitly added by the system to each call. Each transaction is, however, independently handled at each server. That is, the servers neither communicate among themselves nor rely on a centralized component to make scheduling or recovery decisions. In this way, components can be dynamically added or removed from the system without compromising correctness. All a new server needs to know is the interface and address of the servers it will invoke. Regardless of the configuration, the system according to the invention guarantees that transactions executed over these servers will be correct (serializable) and recoverable at a global and local level.

Detailed Description Text (15):
Implementation of locking. Each incoming request to a server is mapped to a thread. In a Java implementation, since these are Remote Method Invocation invocations, this mapping-to-a-thread happens automatically. Setting the corresponding call level lock is done by the thread by creating an entry in a local lock table. If there is no conflicting lock, the thread proceeds to execute the code implementing the service. Otherwise, the thread returns with an exception (implying rollback of the local transaction). By immediately returning an exception, we force the client to be programmed in such as way so as to take into account that an invocation might

not succeed on the first try. On the other hand, resources are more readily available and allow ongoing transactions to terminate sooner.

Detailed Description Text (20):
Implementation of Atomicity. A global transaction is committed using a cascaded variant of 2PC (two-phase commit): each server assumes the role of coordinator for all servers it invokes. To speed up the process, different servers are contacted in parallel: each communication round in the two-phase commit protocol is implemented by one separate thread per server involved. The two-phase commit protocol uses the root identifier as the label to indicate to each server which subtransactions are to be committed. Just like all other communications in the system according to the invention, 2PC happens through RMI. This solves problems with firewalls, because RMI calls can automatically be tunneled through http (hypertext transfer protocol). A negative acknowledgment (a "NO" vote) is implemented as a RemoteException being thrown.

Detailed Description Text (64):
*@exception UnavailableException If the calling thread has no privileges.

Detailed Description Text (90):
*@exception UnavailableException If the calling thread has no *privileges.

First Hit Fwd Refs

End of Result Set

☐ Generate Collection Print

L15: Entry 1 of 1 File: USPT Dec 30, 2003

DOCUMENT-IDENTIFIER: US 6671686 B2
TITLE: Decentralized, distributed internet data management

Abstract Text (1):
A light-weight architecture is provided, where each component is in itself its own advanced mini-transaction processing monitor. To accomplish this, the system is most readily implemented as a set of Java classes. The resulting architecture is as follows. In a composite system, each server is an independent component performing its own scheduling and transaction management. These servers are built using Java and inheriting from the classes provided by the system according to the invention. The interface to each server defines the services it implements. An invocation of one of these services (through remote method invocation) results in the creation of a local transaction (child of the invoking transaction and parent of any transaction that might be triggered by invoking the services of other servers). Each transaction is a thread that (in an exemplary system) can invoke SQL statements in a local database (directly connected to that server) as well as services offered by other servers. All the information that is required to build a global composite transaction is implicitly added by the system to each call. Each transaction is, however, independently handled at each server. That is, the servers neither communicate among themselves nor rely on a centralized component to make scheduling or recovery decisions. In this way, components can be dynamically added or removed from the system without compromising correctness. All a new server needs to know is the interface and address of the servers it will invoke. Regardless of the configuration, the system according to the invention guarantees that transactions executed over these servers will be correct (serializable) and recoverable at a global and local level.

Brief Summary Text (35):
Enterprise Java Beans. This is the Java vision for distributed transaction processing applications. Enterprise Java Beans is a standard, meaning that it consists of specifications rather than implementations. The main objective is to provide a portable way of writing transactional applications. By taking all server-specific issues out of the application (such as transaction management, pooling of resources, swapping of inactive components), it is possible to create portable applications (so-called Beans, the Java terminology for a software component). The key idea is that all these components have to adhere to a standardized way of interacting with the server environment. In practice, this means that a component has a set of predefined methods that are called by the server in case of important events. For instance, before swapping out an inactive component, this component is notified by calling its method ejbPassivate(), whose implementation should discard any volatile data and synchronize the component's database state. The whole concept of this technology is thus oriented towards component-based server applications, and the contract between a component and the server can be very complex. As such, it is orthogonal to the objectives discussed here: although EJB mainly targets applications with transactional aspects, the issue of transaction management itself is left open. In that approach, some of the problems with distributed transactions are recognized, but no attempts are made to solve them. Finally, with JavaBeans, nested transactions are not currently supported.

Brief Summary Text (42):
In the system according to the invention, a light-weight architecture is provided, where each component is in itself its own advanced mini-transaction processing monitor. To accomplish this, the system is most readily implemented as a set of Java classes. The resulting architecture is as follows. In a composite system (such as that of FIG. 2), each server is an independent component performing its own scheduling and transaction management. These servers are built using Java and inheriting from the classes provided by the system according to the invention. The interface to each server defines the services it implements. An invocation of one of these services (through RMI or "remote method invocation") results in the creation of a local transaction (child of the invoking transaction and parent of any transaction that might be triggered by invoking the services of other servers). Each transaction is a thread that can, for example, invoke SQL statements in a local database (directly connected to that server) as well as services offered by other servers. All the information that is required to build a global composite transaction is implicitly added by the system to each call. Each transaction is, however, independently handled at each server. That is, the servers neither communicate among themselves nor rely on a centralized component to make scheduling or recovery decisions. In this way, components can be dynamically added or removed from the system without compromising correctness. All a new server needs to know is the interface and address of the servers it will invoke. Regardless of the configuration, the system according to the invention guarantees that transactions executed over these servers will be correct (serializable) and recoverable at a global and local level.

Detailed Description Text (2):
As described above, in the system according to the invention, a light-weight architecture is provided, where each component is in itself its own advanced mini-transaction processing monitor. To accomplish this, the system is most readily implemented as a set of Java classes. The resulting architecture is as follows. In a composite system (such as that of FIG. 2), each server is an independent component performing its own scheduling and transaction management. These servers are built using Java andainheriting from the cl sses provided by the system according to the invention. The interface to each server defines the services it implements. An invocation of one of these services (through RMI or "remote method invocation") results in the creation of a local transaction (child of the invoking transaction and parent of any transaction that might be triggered by invoking the services of other servers). Each transaction is, for example, a thread that can invoke SQL statements in a local database (directly connected to that server) as well as services offered by other servers. All the information that is required to build a global composite transaction is implicitly added by the system to each call. Each transaction is, however, independently handled at each server. That is, the servers neither communicate among themselves nor rely on a centralized component to makb schedu ing or recovery decisions. In this way, components can be dynamically added or removed from the system without compromising correctness. All a new server needs to know is the interface and address of the servers it will invoke. Regardless of the configuration, the system according to the invention guarantees that transactions executed over these servers will be correct (serializable) and recoverable at a global and local level.

Detailed Description Text (15):
Implementation of locking. Each incoming request to a server is mapped to a thread. In a Java implementation, since these are Remote Method Invocation invocations, this mapping-to-a-thread happens automatically. Setting the corresponding call level lock is done by the thread by creating an entry in a local lock table. If there is no conflicting lock, the thread proceeds to execute the code implementing the service. Otherwise, the thread returns with an exception (implying rollback of the local transaction). By immediately returning an exception, we force the client to be programmed in such as way so as to take into account that an invocation might

not succeed on the first try. On the other hand, resources are more readily available and allow ongoing transactions to terminate sooner.

Detailed Description Text (20):
Implementation of Atomicity. A global transaction is committed using a cascaded variant of 2PC (two-phase commit): each server assumes the role of coordinator for all servers it invokes. To speed up the process, different servers are contacted in parallel: each communication round in the two-phase commit protocol is implemented by one separate thread per server involved. The two-phase commit protocol uses the root identifier as the label to indicate to each server which subtransactions are to be committed. Just like all other communications in the system according to the invention, 2PC happens through RMI. This solves problems with firewalls, because RMI calls can automatically be tunneled through http (hypertext transfer protocol). A negative acknowledgment (a "NO" vote) is implemented as a RemoteException being thrown.

Detailed Description Text (64):
*@exception UnavailableException If the calling thread has no privileges.

Detailed Description Text (90):
*@exception UnavailableException If the calling thread has no *privileges.

First Hit Fwd Refs
End of Result Set

Generate Collection Print

L15: Entry 1 of 1 File: USPT Dec 30, 2003

DOCUMENT-IDENTIFIER: US 6671686 B2
TITLE: Decentralized, distributed internet data management

Abstract Text (1):
A light-weight architecture is provided, where each component is in itself its own advanced mini-transaction processing monitor. To accomplish this, the system is most readily implemented as a set of Java classes. The resulting architecture is as follows. In a composite system, each server is an independent component performing its own scheduling and transaction management. These servers are built using Java and inheriting from the classes provided by the system according to the invention. The interface to each server defines the services it implements. An invocation of one of these services (through remote method invocation) results in the creation of a local transaction (child of the invoking transaction and parent of any transaction that might be triggered by invoking the services of other servers). Each transaction is a thread that (in an exemplary system) can invoke SQL statements in a local database (directly connected to that server) as well as services offered by other servers. All the information that is required to build a global composite transaction is implicitly added by the system to each call. Each transaction is, however, independently handled at each server. That is, the servers neither communicate among themselves nor rely on a centralized component to make scheduling or recovery decisions. In this way, components can be dynamically added or removed from the system without compromising correctness. All a new server needs to know is the interface and address of the servers it will invoke. Regardless of the configuration, the system according to the invention guarantees that transactions executed over these servers will be correct (serializable) and recoverable at a global and local level.

Brief Summary Text (35):
Enterprise Java Beans. This is the Java vision for distributed transaction processing applications. Enterprise Java Beans is a standard, meaning that it consists of specifications rather than implementations. The main objective is to provide a portable way of writing transactional applications. By taking all server-specific issues out of the application (such as transaction management, pooling of resources, swapping of inactive components), it is possible to create portable applications (so-called Beans, the Java terminology for a software component). The key idea is that all these components have to adhere to a standardized way of interacting with the server environment. In practice, this means that a component has a set of predefined methods that are called by the server in ase of important events. For instance, before swapping out an inactive component, this component is notified by calling its method ejbPassivate(), whose implementation should discard any volatile data and synchronize the component's database state. The whole concept of this technology is thus oriented towards component-based server applications, and the contract between a component and the server can be very complex. As such, it is orthogonal to the objectives discussed here: although EJB mainly targets applications with transactional aspects, the issue of transaction management itself is left open. In that approach, some of the problems with distributed transactions are recognized, but no attempts are made to solve them. Finally, with JavaBeans, nested transactions are not currently supported.

Brief Summary Text (42):
In the system according to the invention, a light-weight architecture is provided, where each component is in itself its own advanced mini-transaction processing monitor. To accomplish this, the system is most readily implemented as a set of Java classes. The resulting architecture is as follows. In a composite system (such as that of FIG. 2), each server is an independent component performing its own scheduling and transaction management. These servers are built using Java and inheriting from the classes provided by the system according to the invention. The interface to each server defines the services it implements. An invocation of one of these ser ices (through RMI or "remote method invocation") results in the creation of a local transaction (child of the invoking transaction and parent of any transaction that might be triggered by invoking the services of other servers). Each transaction is a thread that can, for example, invoke SQL statements in a local database (directly connected to that server) as well as services offered by other servers. All the information that is required to build a global composite transaction is implicitly added by the system to each call. Each transaction is, however, independently handled at each server. That is, the servers neither communicate among themselves nor rely on a centralized component to make scheduling or recovery decisions. In this way, components can be dynamically added or removed from the system without compromising correctness. All a new server needs to know is the interface and address of the servers it will invoke. Regardless of the configuration, the system according to the invention guarantees that transactions executed over these servers will be correct (serializable) and recoverable at a global and local level.

Detailed Description Text (2):
As described above, in the system according to the invention, a light-weight architecture is provided, where each component is in itself its own advanced mini-transaction processing monitor. To accomplish this, the system is most readily implemented as a set of Java classes. The resulting architecture is as follows. In a composite system (such as that of FIG. 2), each server is an independent component performing its own scheduling and transaction management. These servers are built using Java and inheriting from the classes provided by the system according to the invention. The interface to each server defines the services it implements. An invocation of one of these services (through RMI or "remote method invocation") results in the creation of a local transaction (child of the invoking transaction and parent of any transaction that might be triggered by invoking the services of other servers). Each transaction is, for example, a thread that can invoke SQL statements in a local database (directly connected to that server) as well as services offered by other servers. All the information that is required to build a global composite transaction is implicitly added by the system to each call. Each transaction is, however, independently handled at each server. That is, the servers neither communicate among themselves nor rely on a centralized component to make schedu ing or recovery decisions. In this way, components can be dynamically added or removed from the system without compromising correctness. All a new server needs to know is the interface and address of the servers it will invoke. Regardless of the configuration, the system according to the invention guarantees that transactions executed over these servers will be correct (serializable) and recoverable at a global and local level.

Detailed Description Text (15):
Implementation of locking. Each incoming request to a server is mapped to a thread. In a Java implementation, since these are Remote Method Invocation invocations, this mapping-to-a-thread happens automatically. Setting the corresponding call level lock is done by the thread by creating an entry in a local lock table. If t ere is no conflicting lock, the thread proceeds to execute the code implementing the service. Otherwise, the thread returns with an exception (implying rollback of the local transaction). By immediately returning an exception, we force the client to be programmed in such as way so as to take into account that an invocation might

not succeed on the first try. On the other hand, resources are more readily available and allow ongoing transactions to terminate sooner.

Detailed Description Text (20):
Implementation of Atomicity. A global transaction is committed using a cascaded variant of 2PC (two-phase commit): each server assumes the role of coordinator for all servers it invokes. To speed up the process, different servers are contacted in parallel: each communication round in the two-phase commit protocol is implemented by one separate thread per server involved. The two-phase commit protocol uses the root identifier as the label to indicate to each server which subtransactions are to be committed. Just like all other communications in the system according to the invention, 2PC happens through RMI. This solves problems with firewalls, because RMI calls can automatically be tunneled through http (hypertext transfer protocol). A negative acknowledgment (a "NO" vote) is implemented as a RemoteException being thrown.

Detailed Description Text (64):
*@exception UnavailableException If the calling thread has no privileges.

Detailed Description Text (90):
*@exception UnavailableException If the calling thread has no *privileges.